

# An Aspect-Oriented Approach for Disaster Prevention Simulation Workflows on Supercomputers, Clusters, and Grids

Tudor B. Ionescu, Andreas Piater,  
Walter Scheuermann, Eckart Laurien  
*Institute for Nuclear Technology and Energy Systems*  
*Universität Stuttgart*  
*70569 Stuttgart, Germany*  
{*ionescu,piater,scheuermann,laurien*}@ike.uni-stuttgart.de

Alexandru Iosup  
*Department of Software Technology*  
*Delft University of Technology*  
*2628 CD, Delft, The Netherlands*  
*a.iosup@tudelft.nl*

**Abstract**—Computer simulation is an important factor in today’s disaster prevention procedures. Simulation codes assess the evolution and impact of various physical phenomena in domains such as nuclear and environmental sciences, and ultimately help saving lives. However, new and more computationally demanding models, and new regulations for personnel training have increased the demand for computational power. While the existing simulation codes can be ported to computing environments that can meet the new demand, such as supercomputers, clusters, and grids, it is too expensive and time-consuming to rewrite and re-certify them. Instead, in this work we propose an aspect-oriented approach that takes existing simulation functionality and combines it with functionality required to run the simulation on different computing environments transparently to the simulation developer. Through experiments in the DAS-3 multi-cluster grid we show that our approach increases the reusability, the maintainability, the scalability, and the robustness of a real disaster prevention simulation, while incurring a low performance overhead.

## I. INTRODUCTION

Disaster prevention procedures rely on time and mission-critical computer simulation. To comply with strict government regulations in areas such as nuclear and environmental sciences, the simulation codes must be updated with the latest research findings, yet still meet time and mission constraints. While computing environments such as supercomputers, clusters, and grids could meet the increasing demand, the simulation codes pose numerous legacy and certification problems that prevent the trivial adoption of these technologies. Since the simulation codes can date back from the late 1970s, they cannot be executed straightforwardly on larger, distributed computing environments. Depending on the mission, the same simulation can run with different requirements such as result accuracy and maximal execution time. The current approach to this problem, custom batch scripts that glue together simulation code and execution policies, is too expensive to develop and quickly becomes unmaintainable. In contrast, in this work we propose an aspect-oriented approach to make disaster prevention simulation more maintainable, scalable, and robust.

To be useful, simulation codes are *integrated into scientific workflows* [1], where each simulation code represents a part of a complex simulation model. Many of the codes have evolved over long periods of time and are written in FORTRAN or C, which poses legacy problems to their integration. Moreover, the workflow can be put in production only after passing a rigorous *certification process*. Rewriting these codes using a modern programming language would be difficult and expensive, especially at universities and research institutes where key knowledge comprised in and around the implementation of the simulation codes has been lost, often due to fluctuations in personnel. Instead, scientists tend to use batch files, shell scripts, and other general-purpose scripting languages to create workflows for their own use and automate the execution of a chain of simulation codes [1]; this has become a common (although bad) practice in research institutes, for rapid prototyping and small projects. However, as the prototypes evolve into long-term projects they become unmaintainable, and the overheads associated with adapting and re-certifying them for new requirements become too costly.

The disaster prevention process also poses quality of service and scalability problems to the simulation codes. The simulation workflows can be executed for different purposes, such as response to a real alarm, personnel training, and experimental tuning of the model; this leads to different computational requirements and governmental rules for each purpose. Due to the evolution of the disaster prevention process, both the number of users and the frequency of using the system increase over time, which leads to scalability issues. Driven by advances in computing environments and by the financial and organizational reality, it has become attractive to port the simulation codes to supercomputers, clusters, and grids, simultaneously, that is, to make the simulation codes *multi-environment executable*. However, the software engineering skills required to port the simulation codes while addressing deployment, persistence, fault-tolerance, access control, etc are often lacking in research institutes, where the number of skilled developers is often much smaller than

the number of scientists with little programming experience.

To solve the tool integration and the multi-environment job execution problems, we propose in this work an aspect-oriented [2] approach for the development of simulation applications. In our approach, developers focus on their core area of expertise, that is, experienced developers on software engineering and system concerns, and scientists with some development skills on model implementation. Thus, aspect-oriented programming (AOP) improves the design, the code, and the modularity of the application while increasing the productivity of developers. Then, the system and the simulation functionalities are combined according to AOP rules, transparently to the developers. The use of AOP allows the legacy simulation codes to run as an integrated tool in different environments. Our main contribution is twofold:

- 1) We investigate the use of AOP for disaster prevention simulation (Section III); our approach elegantly solves the problems of modularity, maintainability, fault-tolerance, and scalability for simulation applications with complex requirements, and also reduces the complexity of the certification process.
- 2) We show the benefits of AOP by comparing an AOP and a conventional implementation of a real application (Section IV); our experiments are carried out in DAS-3, a real multi-cluster grid environment.

## II. A MOTIVATING EXAMPLE: THE ABR SYSTEM

Our motivating example ABR, a real simulation system for disaster prevention based on dispersion calculations<sup>1</sup> for radioactive pollutants. This application is a part of a broader distributed simulation system for remote monitoring of nuclear powerplants. In case of accidents the simulations are time and mission critical within limits and regulations defined by law. In these situations the simulation must be performed in real time, i.e. it uses current weather and emission data and provides results every 10 minutes.

### A. An Overview of the ABR System

The ABR system automates every step of the nuclear disaster prevention process, from data acquisition to reaction to alarms. The system collects periodically radioactive emission and weather data. Emission data, that is, information about the quantity and nature of the released radioactive pollutants are collected at the beginning of each new period (time step). Each sensor information is provided as one of twenty possible incident categories which range from no radioactive emission (cat. 20) to a catastrophic reactor core meltdown (cat. 1). Weather data such as wind and precipitation conditions are fetched from the database of the National

<sup>1</sup>Dispersion modeling is a discipline that provides the mathematical models to calculate the concentration of a substance present in the atmosphere that was released by some source of pollution in any point of an area surrounding it.

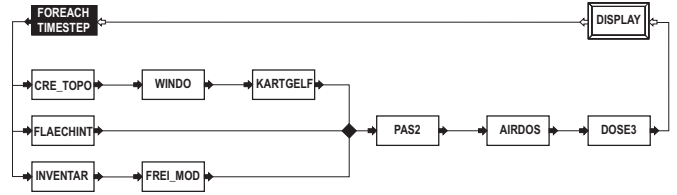


Figure 1. The workflow used to perform dispersion calculations in the ABR system. The functions of the nine existing C++/FORTRAN calculation codes are as follows: *CRE\_TOPO* - generates topographical data on the basis of a homogeneous land surface model; *WINDO* - using the wind forecast data this module computes a 3-dimensional Cartesian wind field through interpolation; *KART\_GELF* - converts the Cartesian wind field into one that takes into account the vertical dimension of land surface; *FLAEC\_INT* - using precipitation forecast data this module computes the distribution of the intensity of precipitation for a given area; *INVENTAR* - computes the nuclide inventory of a nuclear reactor; *FREL\_MOD* - computes the nuclide release from a reactor; *PAS2* - implements a Lagrange dispersion model which uses the input wind field and the distribution of precipitation for the dispersion calculation; *AIRDOS* - computes the equivalent dose of different trace species; *DOSE* - computes the effective dose of radioactivity based on the equivalent dose for different age groups.

Weather Forecast Center for the area surrounding the point of emission under investigation. The system analyzes the evolution and impact of the physical emission process by using a simulation workflow. The results of the simulation are plotted interactively on a digitized geographical map.

The ABR system offers different simulation workflows depending on the required accuracy of the results and computing time. These workflows are based on existing simulation codes written in FORTRAN and C++ which use proprietary data formats. Figure 1 shows a typical simulation workflow based on a Lagrange particle model<sup>2</sup> used to perform dispersion calculations in the ABR system. The simulation workflow is executed once for each time step. The length of a time step is variable and corresponds to the arrival period of the measured values for radioactive emissions (i.e., contaminated gases and aerosols) and weather conditions (e.g., wind speed and direction, precipitations, etc.). After receiving a new set of inputs the state vector of each particle is recomputed according to the new values. A dispersion calculation can be composed of an arbitrarily large number of arbitrarily long times steps.

### B. Present vs. Future Requirements

The ABR system continues to evolve over time in order to comply with new government requirements or to include scientific advances. Table I compares the current status with the planned evolution of the ABR system; the *Future* column refers to the evolution of the system in the following couple of years. Currently there are five main application contexts for the ABR system, two of which (i.e., alarms and serious games) are regulated by law. The *Alarm* context

<sup>2</sup>The government recommendations suggest the use of the Lagrange particle model [3] as the most advanced and realistic dispersion model currently available.

Table I  
PRESENT VS. FUTURE REQUIREMENTS OF THE ABR SYSTEM.

	<i>Present</i>	<i>Future</i>
# of monitored sites*	12 reactors in DE, CH, FR	22 reactors in DE, CH, FR
Monitored area*	r = 25km	r = 100km
# of users	150	250
Time constraints*	10 min / TS	2 min / TS
Application contexts	(1) Alarm*, (2) serious games, (3) <i>What-if</i> scenarios, (4) new model development, (5) certification tests*	Two additional contexts: (6) Periodic calculations for all monitored sited every 1/2 hour* and (7) teaching
HW configuration	two x64 4-cores server	two x64 8-cores server, cluster, grid
Application lifespan	7 years	15 years

\* Regulated by law

represents the scenario of an accident in which case a dispersion calculation is started automatically and must run uninterruptedly for 48 hours. The length of the time step (TS) is 10 minutes and a run of the internal workflow must finish before the arrival of a new measurement dataset. In reality, the calculation must finish in less time so that the simulation process can be restarted in the event of an error. By using recovery points between time steps and by imposing a limit of 2 minutes per time step, the system will be able to restart the calculation up to 4 times if errors occur, which is a performance target for the ABR system. By using a clustered Java Virtual Machine with at least two redundant servers for the alarm calculations, this mechanism currently ensures a satisfying level of fault-tolerance. However, the increase in the number of monitored sites and users and the enlargement of the monitored area around each site will require the system to improve its performance and to scale to over one hundred concurrent calculations for certain application contexts.

In the *serious games* [4], [5] context, false alarms are triggered at nuclear power plants after an irregular time pattern, emulating nuclear accident scenarios. During these alarms, all security measures are taken as for a real accident. The same governmental rules as for the alarm context apply for the serious games context, except that the simulation may not be mission critical. Nevertheless, the same software certification procedure applies to both the alarm and serious games contexts.

In addition to the alarm and serious gaming contexts, the system is used on a regular basis by employees of the ministry of environment (the "*what-if*" scenarios context) and by researchers (the *new model development* and *certification tests* contexts). Additionally, two new application contexts are planned. For reasons of periodical system tests, a dispersion calculation will be started for every half an

hour for every monitored site. It is also planned for the system to be used in a teaching environment. These last five application contexts do not require a certification process as strict as for the alarm and serious gaming contexts. Instead, they require the system to scale to about 100 concurrent calculations in a worst case system load scenario.

Currently, the ABR system operates on a multi-core server and its backup. However, this solution cannot ensure an acceptable performance in terms of computing time for the planned requirements; instead, the use of cluster and grid technology is envisioned.

Last, the prolongation of the application lifespan requires a higher maintainability and reusability of the system and its components, which is where good software engineering is envisioned to help.

### C. Design Goals for the New ABR System

A new simulation engine is needed in order for the ABR system to fulfill all the new requirements for the planned developments. We identify the following design goals for the new simulation engine:

- A clear logical and semantical separation of the distribution, access control, fault tolerance, and persistence cross-cutting concerns from the core functionality;
- The behavior of cross-cutting concerns is to be specified in the application code of the core functionality components through declarative programming;
- The implementation of new features should be easy and will affect a minimal number of system components;
- The elimination of obsolete features must be equally easy to accomplish;
- The implementation must provide a convenient mechanism for activating and deactivating the extended concern-specific functionality in order to achieve a high degree of flexibility through configuration files and better modular testing capabilities;
- The new design must be based on a layered software architecture where communication can take place only between neighbored layers;
- The implementation of concerns like distribution, persistence, etc. must be technology independent and has to be done after the implementation of a solid application core;
- The complete application must be able to cope with various usage contexts (e.g. research, commercial, educational, etc.) and deployment configurations ranging from a multi-server deployment to a notebook installation.

In the following we will focus on describing our aspect-oriented solution for the implementation of the workflows for the ABR system in the spirit of these design goals. Hereby, we especially address the concerns of tool integration, job execution in multiple computation environments, and fault tolerance.

### III. OUR ASPECT-ORIENTED APPROACH

AOP is a relatively new paradigm [2] in computer programming that aims at increasing the modularity and the quality of code. Being based on a correct modular design and good code quality from its inception, a system becomes more maintainable, more flexible, and ultimately requires less code writing, less testing, and less redesign. There exist AOP implementations for several modern programming languages; we use for our own system AspectJ [6], the *de facto* standard AOP language implemented on top of Java.

The foundation of AOP is the *join-point model*. A *join-point* is a particular point in the call graph of a program, such as the execution of a particular method of the program. The value added by the AOP technique is the ability to add functionality to the program *before*, *after*, and/or *instead of* (around the) a join-point, be it a method, class, package, etc. In algebraic terms, AOP exploits the functional decomposition property of computer programs to inject a new function or replace any existing function from it. We call such an added or replaced function *extended functionality*, because it customizes and extends the core functionality of an existing program. The methods implementing extended functionality are called *advices* and reside in a different code module than the core functionality of the system (hence the increase in modularity). The additional behavior is implemented as *aspects* which are ensembles of advices. The advice code is then *woven* into the bytecode of the application at compile-time or run-time, transparently to the developer.

These definitions already suggest that AOP is good for refactoring programs. In addition, our idea is to exploit AOP right from the beginning of the development of a new system by identifying so called *early aspects*. Early aspects are concerns that cross-cut the new system at multiple points whereas the system architects and programmers can foresee that these particular concerns will require writing the same ensemble of code lines (code clones) at many locations in the program. An example of a cross-cutting concern is access-control: each time a call to a specific method occurs, the program control flow must first check that the caller is authorized to perform the requested action. Distribution, logging, persistence, and error handling are other examples of well known cross-cutting concerns that can be foreseen from the start.

#### A. The Scientific Workflow

The simulation codes used by the ABR system are command line executables which communicate with the outside world only through input and output ASCII files. In their basic form, the ABR codes form a heterogeneous system of interconnected modules without a consistent data flow model. The entire workflow must, however, act as a homogeneous model of computation. The Ptolemy II [7] scientific workflow system supports actor oriented hierarchical modeling of heterogeneous systems by focusing on

the data flow, the synchronization of the execution, and the visual design of workflows using the Vergil GUI. The basic building block of a Ptolemy II workflow is the actor - a Java class which, in our case, wraps the FORTRAN and C/C++ codes.

Because Ptolemy II uses the template design pattern, every custom actor class must extend the abstract `TypedAtomicActor` class and is therefore required to have a strict specific structure: it has parameters, I/O ports, and action methods (see Figure 2 (A)). Parameters are set by users; ports are used to interconnect actors, for data flow, and for token based flow control; action methods are invoked by the Ptolemy II workflow manager at different stages of execution, e.g. initialization, fire, wrapup, etc. When the *workflow manager* starts the execution of (fires) an actor, the latter one consumes the tokens on its input ports, performs its job, and produces tokens on its output ports. A *workflow director* dictates the type of interaction and the flow control rules for a particular type of workflow. The most common type of director is the synchronous data flow (SDF) director which uses token based flow control.

The structure of the default Ptolemy II actors has an important drawback for our code development goals: because actors are Java classes with a strict and specific structure the developer either implements all the components of the system as Ptolemy II actors or a Ptolemy II actor wrapper class is needed for each component that has to be integrated into the workflow. The first solution is unacceptable, since components must be reusable in other applications. The second solution generates numerous code clones, which decreases the maintainability of the system. Furthermore, the implementation of Ptolemy II actors requires non-trivial programming skills. Thus, we formulate an additional design goal (see Section II-C): to eliminate the need of implementing a new actor wrapper class for each component of the workflow.

#### B. The Conventional Implementation (Figure 2 (A))

In this first implementation, the ABR codes are encapsulated into Ptolemy II actors (see Figure 2 (A)) and linked together into workflows which are encoded and stored as XML files. For each code the following phases are needed:

- *input preparation* - collects or generates input files for the code;
- *process launch* - launches a local or grid process using the executable code;
- *wrapup and reporting* - parses the output files and stores relevant results produced by the code into a central database.

First, the Ptolemy II engine initializes the parameters using the values given by the designer of the workflow. The input preparation tasks are performed in the `preFire` action method which overrides the method with the same name of the inherited `TypedAtomicActor`. The process

## A. Conventional Implementation

```

public class SimcodeWrapperActor extends TypedAtomicActor {
    Parameter codeExecutable, workingDir, commandArgs, jobTarget, (...);
    TypedIOPort output, input;
    // other field declarations (standard Java types)

    public SimcodeWrapperActor(CompositeEntity container, String name)
        throws NameDuplicationException, IllegalActionException, IOException {
        codeExecutable = new Parameter(...);
        output = new TypedIOPort(this, "baseDirOut", false, true);
        output.setTypeEquals(BaseType.OBJECT);
        // other port and parameter initializations
    }

    @Override
    public boolean prefire() throws IllegalActionException {
        // prepare input files for simcode
        return success;
    }

    @Override
    public void fire() throws IllegalActionException {
        // fetching values from parameters and input ports
        try {
            if (sJobTarget.compareTo("GRID")==0) {
                JobDispatcher.dispatchGridJob(...);
            } else if (sJobTarget.compareTo("CLUSTER")==0) {
                JobDispatcher.dispatchClusterJob(...);
            } else if (sJobTarget.compareTo("LOCAL")==0) {
                JobDispatcher.dispatchLocalJob(...);
            }
        } catch (JobDispatchException e) {
            // this exception must be handled here
        }
    }

    @Override
    public boolean postfire() throws IllegalActionException {
        // process output files from simcode
        // sending tokens on output ports
        return success;
    }
}

```

## B. AOP Implementation

```

public class SimcodeWrapper {
    // field declarations (standard Java types only)
    public SimcodeWrapper() {...} // this constructor is required

    @ActorParameter(name = "jobTarget")
    public void setJobTarget(String jobTarget) {
        this.jobTarget = jobTarget;
    }
    @ActorParameter(name = "codeExecutable")
    public void setCodeExecutable(String codeExecutable) {
        this.codeExecutable = codeExecutable;
    }
    // other annotated setter methods

    @ManagedJobRemoteCommand
    public String getCodeExecutable() { return codeExecutable; }

    @ManagedJobTarget
    public String getJobTarget() { return jobTarget; }
    // other annotated getter methods

    @ActorAction(actionMethods = {ActionType.Prefire}, callingOrder = 0)
    public void prepareInput(
        @ActorPort(name = "workingBaseDir", direction = PortDirection.Input)
        String workingBaseDir) {
        this.workingBaseDir = workingBaseDir; // received on the ActorPort
        // prepare input files for simcode
    }

    @ManagedJob(proceed=true)
    @ActorAction(actionMethods = {ActionType.Fire}, callingOrder = 0)
    public void runSimcode() throws JobDispatchException {
        JobDispatcher.dispatchLocalJob(...);
    }

    @ActorAction(actionMethods = {ActionType.Postfire}, callingOrder = 0)
    @ActorPort(name = "outputToken", direction = PortDirection.Output)
    public Object processOutput() throws Exception {
        // process output files from simcode
        return workingBaseDir; // will be sent on the declared ActorPort
    }
}

```

Figure 2. A conventional vs. an AOP implementation of a Ptolemy II actor.

is launched in the `fire` action method where the decision upon the target execution environment is taken. In this example we have three options: local, cluster and grid execution. The `JobDispatcher` class implements simple retry and redundancy mechanisms are implemented to provide basic fault tolerance if the process launch fails from reasons that can be traced back to errors that do not follow a regular pattern (e.g., a cluster or grid job submission fails due to a hardware failure at a specific node; upon the next submission the scheduler avoids sending jobs to that particular node). Finally, in the `postFire` action method the wrapup and reporting tasks are performed and a token is released for the next actor to be able to fire.

In this example, we have two types of code elements that negatively affect the maintainability and reusability of this component: Design pattern specific code and API calls. The explicit `TypedIOPort`, `Parameter`, and action methods declarations are only relevant in the context of the Ptolemy II system. If we were to migrate the application to another technology, say a different workflow engine, we would have to rewrite this component entirely. Furthermore explicit API calls, as the ones in the `fire` action method pose maintainance and certification problems. From the maintainability point of view, each time the job execution API changes (e.g., when a new version of the API is released), all calls to that API have to be reviewed and (possibly) replaced or extended

with other calls since they represent code clones that can be found in all classes of this type. From the certification point of view, this code is not optimal because it involves a technology (i.e., cluster/grid execution) which is costly and time-consuming to certify for the two usage contexts of the ABR system: alarm calculations and serious gaming.

## C. AOP Implementation of the Cross-Cutting Concerns (Figure 2 (B))

We now employ a different, AOP-based implementation strategy.

1) *Tool Integration*: In the AOP-based strategy we start with a *plain old java object (POJO)* which does not contain any calls to complex external APIs nor does it contain API-specific declarations. The only restriction imposed to it is the Java beans convention regarding getter and setter methods.

To transform this POJO into an actor class we use Java Annotations. This is a standard feature of the Java language that allows adding custom modifiers that precede package, class, method, or field declarations. Annotations are not processed by the Java compiler the same way as other language constructs in the sense that, by default, they have no predicative effect. Their purpose is purely declarative and in order to access the information provided by annotations reflective programming has to be employed. In other words, there must be a component that analyzes the objects of a

class, reads the annotations that decorate this class and its elements, and finally takes an action using the information provided by the annotations and the features of the object. Figure 2 (B) shows the annotated version of the actor class first introduced in Figure 2 (A). The difference is that in the AOP version the API-specific *declarations and calls* have been replaced by API-specific *annotations*. Using AOP and reflection, two aspects called `ActorAspect` and `GridAspect` are activated whenever needed; note that neither of the two aspects is shown in Figure 2 (B), which emphasizes the separation of aspects from code code. The aspect then automatically performs all API-specific initializations and calls that allow this class to act as a Ptolemy II actor and to virtualize the execution environment.

Table II (A) shows the three annotations that are needed to automatically integrate a POJO with the workflow engine. This mechanism works as follows:

1. Ptolemy II reads an XML workflow and for each atomic actor it initializes an actor object of a generic actor type we defined, called `ActorBase`, which contains an empty list of ports and parameters; when designing the workflow we have renamed this actor to reflect the name of the annotated POJO class.

2. The `ActorAspect` reacts whenever an object of the type `ActorBase` is instantiated and uses its name to instantiate the annotated POJO using reflection; it then creates the ports and parameters specified using the respective annotations.

3. Finally, the `ActorAspect` captures any call to an action method of `ActorBase` and uses reflection to Figure out which POJO method to call; in particular, it uses the information provided by the `actionMethods` field of the `ActorAction` annotation.

2) *Job Execution*: The `GridAspect` introduced by the AOP-based strategy works slightly different in that it uses a pointcut defined on the basis of the presence of the `@ManagedJob` annotation over a method. Table II (B) shows the annotations that are used for automatic job target environment selection and execution. Some of these annotations represent a mapping of the DRMAA `JobTemplate` class which defines all fields that are relevant for job execution in grid environments and clusters. The `GridAspect` searches for annotated getters of the POJO and extracts information like job executable (method `getCodeExecutable` in Figure 2 (B)), working directory, input path, etc. and passes it to the `JobDispatcher`. This information is used by the `JobDispatcher` to perform the actual job submission and to provide the four fault tolerance mechanisms which will be described in the next subsection.

In the current implementation, if the `@ManagedJobTarget` is set to `LOCAL` the job is executed on the local machine using the `Java Runtime.exec()` method; if it is set to `CLUSTER` the job is submitted using

Table II  
ANNOTATIONS FOR TOOL INTEGRATION AND JOB EXECUTION.

A.) Annotations for Tool Integration	
Annotation	Effect
<code>@ActorPort</code>	Over a method: states that the return value of the method is to be encapsulated into a token and put on an output port. Over an input argument of a method: states that the value of an argument is received on an input port of the actor.
<code>@ActorParameter</code>	Over a setter: indicates that the value which is set represents a definable Ptolemy II actor parameter.
<code>@ActorAction</code>	Over a method: indicates the POJO method to be called when the action method of the Ptolemy II actor class specified by the <code>actionMethods</code> annotation parameter is invoked by the workflow manager.
B.) Annotations for Job Execution Management	
Annotation	Effect
<code>@ManagedJob</code>	Over a method: indicates that a call to this method triggers a cluster or a Grid job.
<code>@ManagedJobTarget</code>	Over a getter: indicates the target execution environment. The possible values are: <code>LOCAL</code> , <code>CLUSTER</code> , and <code>GRID</code> .
<code>@ManagedJobRetries</code>	Over a getter: states the number of submission attempts before proceeding with the execution of the POJO method.
<code>@ManagedJobTimeout</code>	Over a getter: the number of milliseconds to wait before aborting and retrying the job submission.
<code>@ManagedJobRedundancy</code>	Over a getter: states the number of redundant job submissions.
<code>@ManagedJobQueueCapacity</code>	Over a getter: the number of concurrent job requests that can be queued for submission.
<code>@ManagedJobRetryPeriod</code>	Over a getter: the delay between two retry attempts.
<code>@ManagedJobParam</code>	Over a getter: indicates a DRMAA job template parameter. <i>Param</i> can take the values: <i>RemoteCommand</i> , <i>Args</i> , <i>Error-Path</i> , etc.

DRMAA and the Sun Grid Engine (SGE) cluster scheduler; if it is set to `GRID` the job is submitted using the `KOALA` grid scheduler [8].

3) *Fault-Tolerance*: We have implemented in the `JobDispatcher` four fault tolerance mechanisms, namely retry, redundancy, fall-back, and job queueing. We now discuss each of them in turn.

The first two fault tolerance mechanisms provided by the job dispatcher are based on redundant submissions (i.e., the job is submitted redundantly to a given number of nodes specified by an annotation and the results of the job that finishes first are considered) and retries (i.e., upon failure submission is retried for a given number of times specified by another annotation).

By using an *around* advice to submit grid or cluster jobs the `GridAspect` uses the *proceed* feature of `AspectJ` around advices to execute the underlying POJO method whenever the job dispatcher of the `GridAspect` reports an error. If for some reason the remote execution fails despite

all retry and redundancy attempts, the fall-back strategy is applied. Then, the `runSimcode` method of the POJO from Figure 2 (B), which launches a local process, is actually executed and thus provides the system with an additional layer of fault tolerance. Furthermore, by not activating the `GridAspect` at all for certain workflow, the same POJO can be used in the alarm and serious gaming application contexts where only the local execution of jobs is permitted.

The job dispatcher implements a job queue that prevents excessive concurrent submission requests. This is useful for grid schedulers like KOALA which do not implement a job queue for atomic jobs. The job queue also prevents the opening of too many files concurrently which is a known problem for Linux and Unix systems with strict policies for process execution.

#### IV. THE VALIDATION OF OUR APPROACH

In this section we validate our AOP-based approach for disaster prevention simulation. Towards this end, we present two types of experiments. First, we validate our approach from a software engineering perspective, by comparing an AOP-based and a conventional implementation of the same production simulation (see Section II). Second, we validate the ability of our approach to operate the real simulation in cluster and multi-cluster grid environments; for this second type of experiments we look not only at scalability, but also at the reliability of the simulations.

##### A. Reusability and Maintainability Analysis

We have compared our AOP approach with a conventional approach of implementing Ptolemy II actors. For the conventional approach we have fully implemented the generic Ptolemy II actor depicted in Figure 2 (A) using only object-oriented code, without AOP. Conversely, for the AOP approach we have fully implemented the actor sketched in Figure 2 (B). Both codes are available online<sup>3</sup>. We have then evaluated both implementation using common software metrics for reusability and maintainability [9], [10]. The results of the comparison are shown in Table III.

One of the factors that affect the reusability and maintainability of programs is the size of the code: the larger the program, the less reusable it is. The *lines of code* metric is defined as the total number of lines of a program or class (comments are not counted). Because annotations represent additional information for programmers and compilers (which can also ignore them) they were considered instead in the *lines of annotations* metric. The AOP implementation leads to about 18% less code and over 10 times more lines of annotations than the conventional one. While a large number of annotations may lead to less understandable and less maintainable code, by using tool support for intelligent grouping and selective displaying of code and annotation

<sup>3</sup>The source code and test workflows are available for download at: <http://code.google.com/p/aosif>

Table III  
RESULTS OF THE COMPARISON OF THE AOP VERSION VS. CONVENTIONAL VERSION OF THE SIMCODEWRAPPER COMPONENT IN TERMS OF DEVELOPMENT TIME AND CODE SIZE.

<i>Metric</i>	<i>AOP Impl.</i>	<i>Conventional Impl.</i>
Lines of code	76	92
Lines of annotations	23	3
Total lines (code + annotations)	99	95
Imports	5	9
External (API) classes used	1	7
Depth of inheritance	0	5
Performed functions	1	2
Development time (min)	35	48
Modification time (min)	8	12
Execution time (sec)	21	20

sections in programs the readability of the code is not affected. In fact, if we were to strip all annotations from the AOP wrapper class, we would be left with a basic and reusable class mostly composed of getters and setters for the class attributes, which are simple to understand even for novice developers.

Inter-class couplings (imports) and calls to API classes (i.e. library classes) are considered to have a negative impact upon reusability and maintainability because they violate the modularity paradigm by introducing invisible non-standard communication interfaces between different modules. The same applies to classes with high values for the depth of the inheritance tree. Looking at the two implementations in Figure 2 one can notice that in the AOP implementation there is only one explicit call to an external class, namely to the `JobDispatcher.dispatchLocalJob` method whereas in the conventional implementations there are numerous calls to Ptolemy II specific classes (e.g. `Parameter`, `TypedIOPort`, etc.).

The number of distinct functions of a method also has an impact on the reusability of the containing class: the number of functions of a method is indirectly proportional to the probability for it to be reusable and should therefore be 1. For example, the `fire` method of the conventional implementation violates this principle because it actually has two functions: launching a simulation code run and deciding which job execution environment to be used. The latter is a cross-cutting concern that should be implemented in a separate module. The AOP implementation does exactly that by leaving the decision up to the `GridAspect`.

A higher degree of maintainability of the code leads to shorter development and modification times which, in term, increase productivity. The values shown in Table III are referring to developing and modifying a simulation code wrapper actor by a programmer familiar with both AOP and Ptolemy II. They show that the AOP approach leads to about 30% less development and modification time.

Finally, because AOP introduces a certain computational burden we also measured the execution time for a one hour

Table IV

THE CHARACTERISTICS OF THE FIVE DAS-3 CLUSTERS. CPU FAMILY = AMD OPTERON. OS = SCIENTIFIC LINUX; CLUSTER SCHEDULER: SUN GRID ENGINE; GRID SCHEDULER: KOALA.

	# of machines	2x CPU	2x Core	Freq. (GHz)
Cluster 1	85	yes	yes	2.4
Cluster 2	32	yes	no	2.6
Cluster 3	41	yes	yes	2.2
Cluster 4	68	yes	no	2.4
Cluster 5	46	yes	no	2.4

long time step using the ABR workflow from Figure 1 for both implementations. The loss in performance is the cost that has to be paid for the increased modularity, but is still acceptable (below 5%). This loss can, however, be reduced by executing longer time steps and other optimizations.

### B. Tests on the DAS-3 Grid

To show that our solution provides the necessary features to port time and mission-critical simulation application to cluster and grid execution environments with little effort, good performance, and a satisfying level of fault tolerance, we have ported the simulation workflow described in Section II, using AOP. Then, we carried out a series of tests on the DAS-3 grid in order to evaluate the applicability and the performances of our AOP components. Table IV summarizes the characteristics of the five clusters forming the DAS-3 grid.

We used the workflow presented in Figure 1 for testing. For simplicity, prefetched weather forecast values have been used in order to avoid connecting to the weather forecast data server. The simulation duration was set to a single one hour-long time step. In Table I we have presented the future requirements for a ten-minutes time step, including a maximal execution time of 2 minutes per time step. This means that each hour-long workflow used in these experiments must finish in under 12 minutes to cope with the new requirements to the system.

The workflow has been ported to use the DRMAA based submission to the SGE cluster scheduler and the regular submission process of the KOALA grid scheduler for cluster and grid execution, respectively. We used only Cluster 4 (having 68 nodes) for the cluster experiments, and all the five DAS-3 clusters for the grid experiments. To stress-test the system, we ran independent sets of simultaneously running workflows. The number of simultaneous workflow runs present in each set ranged from 50 to 200, in increments of 25; 200 is twice the maximum number of concurrent simulations that are expected considering the new system requirements. The fault tolerance parameters for the job dispatcher of the `GridAspect` were set as follows: `redundancy = 1` (no redundancy), `retries = 10` (if any of the workflow jobs fails more than 10 times, the whole workflow is aborted), `queueCapacity = 200` jobs,

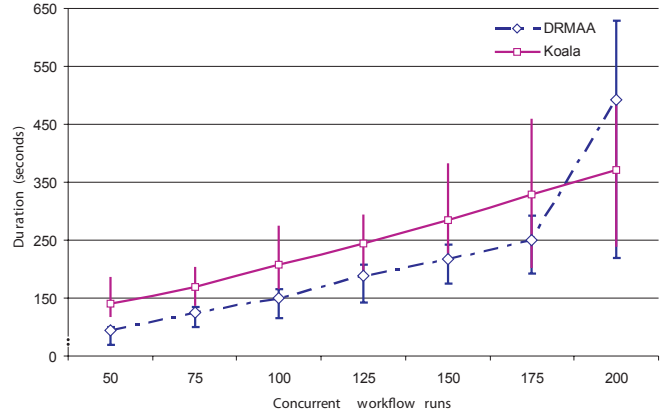


Figure 3. Workflow duration when running on the cluster (through a DRMAA interface) and DAS-3 grid (through a KOALA interface). For each experiment size, the point represents the average workflow duration, and the error bars the minimum and maximum workflow duration.

and `jobTimeout = 60` seconds. These values have been selected to be similar with previous tests performed on the DAS system [11].

The tests were performed during normal working hours, when the system load is about 15%, as indicated by long-term studies of the DAS system workloads [12], [13]. The results of the tests are shown in Figure 3. The DRMAA workflow consistently needs a shorter time to finish than the KOALA workflows until 200 workflow runs are submitted concurrently and the cluster becomes too overloaded to cope. For this high overload case, the performance drop was caused by timeout errors of the SGE scheduler and the running time almost reaches the 12 minutes barrier for a workflow. We conclude that when possible the use of a single cluster through its direct resource management interface, DRMAA, is preferable to the use of the multi-cluster grid, where the overheads generated by inter-cluster job scheduling, job submission, and file transfers are significant.

Although all workflows finished successfully, the submission of some of the jobs failed. Table V provides an overview of the number of failed job submissions and the number of job submission retries that were needed to successfully finish the workflows. While for the KOALA workflow there are cases of failure even when 50 concurrent workflow runs are launched, the SGE scheduler shows its limitations only when the number of concurrent workflow runs reaches and exceeds 175.

Table V

THE NUMBER OF JOBS REQUIRING ONE/MORE THAN ONE RETRY ATTEMPTS TO COMPLETE. EACH WORKFLOW CONSISTS OF 8 JOBS.

# of workflows	50	75	100	125	150	175	200
DRMAA	0/0	0/0	0/0	0/0	0/0	37/0	140/137
KOALA	9/0	8/0	14/0	16/0	39/0	139/21	136/24

Overall, the results of the tests show that the AOP implementation works and that the simple fault tolerance features of the `GridAspect` are effective in the sense that all workflows finish. Nevertheless, this also suggests the need for quality of service features that ensure a fixed time limit for the execution of a workflow. Performance- and reliability-wise, the results also show that the cluster execution environment is reliable when the number of concurrent jobs does not greatly exceed the number of nodes present in the cluster. Thus, the cluster can be considered a more scalable alternative to the multi-core server solution for all the application contexts of the ABR system. By using reliable fault tolerance mechanism, the grid environment can be used for the application contexts of the ABR system which are not mission critical, and when the local resources are overloaded. We conclude that the multitude of available options concerning the job execution environment and the implementation technologies justifies our AOP approach and proves one of its main advantages: it provides the necessary flexibility for deploying many versions of a single system which can be customized through configuration parameters rather than needing different versions of the application code.

## V. RELATED WORK

Our work stands at the intersection between software engineering and large-scale distributed simulation. We now review the relevant related work from these two areas, in turn. Previously, AOP has been used for decoupling from the core software architecture cross-cutting concerns such as quality of service [14], automated software updates [15], and fault tolerance [15], [16], [14]. (Security [17] and persistence [18] annotations are already standardized Java 5.0 features.) AOP has been previously used by the developers of the Ptolemy II system [7] to implement the backtracking fault tolerance mechanism for workflows. Extending previous work, ours is the first to use an AOP-based approach for (large-scale) distributed simulations.

Large-scale distributed simulation has received much attention in the past decade. The Cactus [19] and the Discover [20] projects propose each a framework that encapsulates and manages (among others) the distributed simulation. The projects Condor Master-Worker [21], DIRAC [22], and BOINC [23] are frameworks for the execution of generic applications on large-scale distributed environments; for BOINC the resources are provided by volunteers that may not be associated with the application developers. The Aurora2 [24] project optimizes distributed simulations on volunteered resources; our work has a different focus, and the law does not permit the disaster prevention simulations to run on volunteer resources. For all these approaches, the user has to explicitly call in the simulation code the API of the framework, which makes the code development and certification difficult for our domain (see Section II).

Orthogonal to our work is the standardization of the interfaces for interconnecting simulation applications, a problem which has been addressed by the High Level Architecture (HLA) specification for simulation applications [25] and its updates. The federates model of the HLA specification allows for the coupling of different remote simulation applications developed by different organizations by implementing a federate interface. However, HLA is often perceived as a "heavy" standard [26], that is, complex, and difficult to learn and adopt. Moreover, despite much work put into improving HLA functionality for large-scale distributed environments, especially grids [27], there are still many challenges to be addressed [28]. In contrast to HLA, we addressed the issue of reusability within a single organization where non standard interfaces are used to couple the components of a single simulation application.

## VI. CONCLUSION AND FUTURE WORK

Disaster prevention simulation raises new challenges in tool integration and multi-environment execution. To address these challenges, in this paper we have presented a new aspect approach for developing reusable workflows in distributed mission critical simulation systems. Starting from a list of design goals with emphasis on a clear logical and semantical separation of cross-cutting concerns from the core functionality of the system, we showed how our AOP based solution leads to a higher degree of flexibility in choosing the implementation technology for tool integration and multi-environment job execution. In our approach, the desired extended behaviour of the system is expressed in the application code of the core simulation application through declarative programming, thus providing a flexible yet technology-neutral interface between the core and the extended functionality. The link between these two distinct functionalities is realized through aspects, which gives the developer complete freedom concerning the implementation of the extended functionality. This results in an increased maintainability and reusability of the application code. To validate our approach, we used the example of a distributed simulation system for the remote monitoring of nuclear powerplants. Our code analysis showed that, compared to the conventional solution, the AOP solution significantly reduces the size of the code and the development time with an acceptable performance overhead. Through experiments in the real DAS-3 grid we also showed that the simulation can run on cluster and grid environments scalably and reliably.

Overall, we found that aspect-oriented approach is applicable for disaster prevention simulation, and offers many advantages over the more conventional programming paradigms. As part of our future work we consider extending our investigation to other implementation concerns like distribution, persistence, access control, and quality of service, for which we plan to provide a consistent collection of annotations and aspects as a thin programming library. We

also plan to investigate the use of the new cloud computing environments, where resources are provided for a cost but come with performance and reliability guarantees.

## VII. ACKNOWLEDGEMENTS

This research has been supported by the Ministry of Environment of the German State of Baden-Württemberg. Access to the DAS-3 grid has been kindly granted by the Delft University of Technology.

## REFERENCES

- [1] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher, "Scientific workflow design for mere mortals," *Future Gener. Comput. Syst.*, vol. 25, no. 5, pp. 541–551, 2009.
- [2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP '97 - Object-Oriented Programming*. Springer-Verlag Berlin, 1997, pp. 220–242.
- [3] S. Raza and R. Avila, "A 3d lagrangian particle model for direct plume gamma dose rate calculations," *Journal of Radiological Protection*, vol. 21, pp. 145–154(10), 2001.
- [4] H. Kelly, K. Howell, E. Glinert, L. Holding, C. Swain, A. Burrowbridge, and M. Roper, "How to build serious games," *Commun. ACM*, vol. 50, no. 7, pp. 44–49, 2007.
- [5] D. Michael and S. Chen, *Serious Games: Games That Educate, Train, and Inform*. Berkely, CA, USA: Course Technology PTR, 2005.
- [6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2001, pp. 327–353.
- [7] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity the ptolemy approach," in *Proceedings of the IEEE*, vol. 91, 2003, pp. 127–144.
- [8] H. H. Mohamed and D. H. J. Epema, "Koala: a co-allocating grid scheduler," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 16, pp. 1851–1876, 2008.
- [9] J. Barnard, "A new reusability metric for object-oriented software," *Software Quality Control*, vol. 7, no. 1, pp. 35–50, 1998. [Online]. Available: <http://dx.doi.org/10.1023/B:SQJO.0000042058.34876.c8>
- [10] Y. Lee and K. H. Chang, "Reusability and maintainability metrics for object-oriented software," in *ACM-SE 38: Proceedings of the 38th annual on Southeast regional conference*. New York, NY, USA: ACM, 2000, pp. 88–94.
- [11] A. Iosup and D. H. J. Epema, "Grenchmark: A framework for analyzing, testing, and comparing grids," in *CCGRID*, 2006, pp. 313–320.
- [12] H. Li, D. L. Groep, and L. Wolters, "Workload characteristics of a multi-cluster supercomputer," in *JSSPP*, 2004, pp. 176–193.
- [13] A. Iosup, C. Dumitrescu, D. H. J. Epema, H. Li, and L. Wolters, "How are real grids used? the analysis of four grid traces and its implications," in *GRID*, 2006, pp. 262–269.
- [14] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky, "Building adaptive distributed applications with middleware and aspects," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2004, pp. 66–73.
- [15] S. Fleissner and E. L. A. Baniassad, "Epi-aspects: aspect-oriented conscientious software," *SIGPLAN Not.*, vol. 42, no. 10, pp. 659–674, 2007.
- [16] S. Bouchenak, N. D. Palma, S. Fontaine, and B. Tête, "Aosd for internet service clusters: the case of availability," in *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*. New York, NY, USA: ACM, 2005.
- [17] S. Microsystems, "Security annotations and authorization in glassfish and the java ee 5 sdk," 2006.
- [18] C. Bauer and G. King, *Hibernate in Action (In Action series)*. Greenwich, CT, USA: Manning Publications Co., 2004.
- [19] G. Allen, W. Bengler, T. Damlitsch, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf, "Cactus tools for grid applications," *Cluster Computing*, vol. 4, no. 3, pp. 179–188, 2001.
- [20] H. Liu, L. Jiang, M. Parashar, and D. Silver, "Rule-based visualization in the discover computational steering laboratory," *Future Generation Comp. Syst.*, vol. 21, no. 1, pp. 53–59, 2005.
- [21] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience," *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.
- [22] A. Tsaregorodtsev, V. Garonne, and I. Stokes-Rees, "Dirac: A scalable lightweight architecture for high throughput computing," in *GRID*, 2004, pp. 19–25.
- [23] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *GRID*, 2004, pp. 4–10.
- [24] A. Park and R. Fujimoto, "A scalable framework for parallel discrete event simulations on desktop grids," in *GRID*, 2007, pp. 185–192.
- [25] J. S. Dahmann and K. L. Morse, "High level architecture for simulation: An update," in *DIS-RT '98: Proceedings of the Second International Workshop on Distributed Interactive Simulation and Real-Time Applications*. Washington, DC, USA: IEEE Computer Society, 1998, p. 32.
- [26] S. Straßburger, T. Schulze, and R. Fujimoto, "Future trends in distributed simulation and distributed virtual environments: Results of a peer study," in *Winter Simulation Conference*, 2008, pp. 777–785.
- [27] K. Pan, S. J. Turner, W. Cai, and Z. Li, "A service oriented hla rti on the grid," in *IEEE International Conference on Web Services (ICWS 2007)*, 2007, pp. 984–992.
- [28] S. J. Turner, "Distributed simulation on the grid: Opportunities and challenges," in *DS-RT*, 2005, p. 1.